Performance Test Case Generation for Java and WSDL-based Web Services from MARTE

Antonio García-Domínguez and Inmaculada Medina-Bulo
Department of Computer Languages and Systems
University of Cádiz
Cádiz, Spain
{antonio.garciadominguez, inmaculada.medina}@uca.es

Mariano Marcos-Bárcena
Department of Industrial Design and Mechanical Engineering
University of Cádiz
Cádiz, Spain
mariano.marcos@uca.es

***Abstract*—Obtaining the expected performance from a workflow would be easier if every task included its own specifications. However, normally only global performance requirements are provided, forcing designers to infer individual requirements by hand. Previous work presented two algoritms that automatically inferred local performance constraints in Unified Modelling Language activity diagrams annotated with the Modelling and Analysis of Real-Time and Embedded Systems profile. This work presents an approach to use these annotations to generate performance test cases for multiple technologies, linking a performance model and a design model with a weaving model in a meet-in-the-middle approach so users can write their software according to their needs. Two implementations of the approach are described, which have been published as open source software. The first implementation extracts models from Java unit tests using MoDisco and weaves them with the performance models in order to convert some or all of the test cases in the selected JUnit test suites to performance tests. The second implementation extracts models from WSDL documents including message templates and template variables and uses these models for generating test inputs, test code and test infrastructure. Users can customise the service catalogues, test inputs and message templates to obtain a high degree of flexibility, and the test infrastructure provides automated configuration and test reports. These two successful implementations for different technologies validate the proposed approach as a generic framework for generating performance test case artefacts from existing software.**

Keywords-software performance; Web Services; MARTE; model driven engineering; test generation.

I. Introduction

Software needs to meet both functional and non-functional requirements. Performance requirements are among the most commonly used non-functional requirements, and in some contexts they can be just as important as functional requirements. In addition to soft and hard real-time systems, Service Oriented Architectures (SOAs) must be considered as well. Within SOAs, it is common practice to sign Service Level Agreement (SLAs) with external services, to compensate consumers in case of problems. It is also quite common to create "service compositions", which are services that integrate several lower level services (normally, Web Services from external providers). However, it may be difficult to establish what performance level should be required from the composed services. Too little, and the performance requirements for the composition will not be met. Too much, and the provider may charge more than desired. In addition, developers must test the external services to ensure that they can provide the required performance levels.

There is a large variety of proposals for estimating the required level of performance and measuring the actual performance of a system. Measurements can be used for detecting performance degradations over time, identifying load patterns or checking the SLAs. However, the requirements set by the SLA are usually broad and cover a large amount of functionality: when violated, it might be hard to pinpoint the original cause. Whenever possible, performance requirements should be as specific as possible, but that would be too expensive for all but the most trivial systems.

This paper is an extended version of our previous work [1]. This previous work presented an overall approach for using the models produced by the inference algorithms in [2] to generate performance artefacts for multiple target technologies. The algorithms can "fill in the blanks" for the response time and throughput requirements of every activity in the model, starting from a global annotation and some optional local annotations set by the user. Originally, users would then have to write the actual performance tests manually, taking the results produced by these algorithms as a reference. However, writing these tests for every part of a reasonably-sized system could incur in a considerable cost: ideally, it should be partly automated.

The present work provides an up-to-date account of the results obtained after implementing the two approaches outlined in our previous work. Though the overall approach has not changed much, many details had to be revised as the tools were defined. For instance, the Java approach now offers finer-grained control on the tests to be used and can use other metrics apart from maximum response times, such as averages, medians or percentiles. It is the WSDL-based approach that has changed the most: WSDL documents have been found to be too complex for doing model weaving directly on them. Instead, a custom model extractor has been developed, which is combined with a test generator and a template language to generate the input messages.

The rest of this work is structured as follows. After discussing related work in Section II, the MARTE profile is introduced in Section III and the performance models are presented in Section IV. Section V describes the general

approach for generating test artefacts. This approach is applied to reusing Java unit tests as performance tests using the ContiPerf library in Section VI. Section VII is dedicated to generating performance tests (input data, testing code and infrastructure) from any Web Service described using WSDL. Finally, conclusions and future lines of work are listed in Section VIII.

II. Related work

According to Woodside et al. [3], performance engineering comprises all the activities required to meet performance requirements. These activities include defining the requirements, analysing early performability models (such as layered queuing networks [4] or process algebra specifications [5]) or testing the performance of the actual system. Our previous work in [2] focused on helping the user define the requirements using MARTE-annotated [6] UML activity diagrams as notation. The present work will show how to assist the user in creating the performance test artefacts from the resulting requirements.

Many testing approaches do not work directly with the implemented system, but rather with a simplified representation (a model). There is a large number of works dealing with model-based testing, i.e., "the automatable derivation of concrete test cases from abstract formal models, and their execution" [7]. Most of them (as evidenced by [7] itself) are dedicated to functional testing: the rest of this section will focus on those dedicated to model-based performance testing.

Barna et al. present in [8] a hybrid approach, which uses a 2-layered queuing network (LQN) to derive an initial stress workload for a website. This workload is used to test the system and refine the original LQN model in a feedback loop that searches for the minimum load that would make the system violate one of its performance constraints. Like our work, it combines the analysis of a model with the execution of a set of test cases. However, its goal is completely different: the algorithms in [2] intend to define the appropriate quality service levels for the individual services in order to meet the desired quality service level of the entire workflow, whereas this approach would estimate the maximum workload that a workflow could handle within a certain quality service level.

Di Penta et al. show in [9] another approach with the same goal of finding workloads that induce service level agreement violations. However, they use genetic algorithms instead of a LQN model and test WSDL-based Web Services instead of a regular website.

Suzuki et al. have developed a model-based approach for generating testbeds for Web Services [10]. SLA and behaviour models are used to generate stubs for the external services used by the service. This allows users to check that their own services can work correctly and with the expected level of performance as long as the external services meet their SLAs. However, this approach does not generate input messages for the services themselves. Still, we could use this work to check the validity of the performance constraints inferred by the algorithms in [2] in combination with the approach which will be presented in Section VII, by replacing all services in the workflow with stubs and testing the performance of the composition.

As illustrated by the above references, there is a wealth of methods for generating performance test cases and testbeds for Web Services. However, we have been unable to find another usage of model weaving for generating performance test artefacts for multiple technologies. This is in spite of the fact that model composition using model weaving has been used regularly ever since the authors of the original ATLAS Model Weaver proposed it [11]. For instance, Vara et al. use model composition to decorate their extended use case models with additional information required for a later transformation [12].

III. The MARTE profile

UML is widely used as a general purpose modelling language for software systems. However, UML cannot model non-functional aspects such as performance requirements.

For this reason, the OMG (Object Management Group) proposed in 2005 the SPT (Schedulability, Performability and Time) profile [13], which extended UML with a set of stereotypes describing scenarios that various analysis techniques could take as inputs. In 2008, OMG proposed the QoS/FT (Quality of Service and Fault Tolerance Characteristics and Mechanisms) profile [14], with a broader scope than SPT and a more flexible approach: users formally defined their own quality of service vocabularies to annotate their models.

When UML 2.0 was published, OMG saw the need to update the SPT profile and harmonise it with other new concepts. This resulted in the MARTE (Modelling and Analysis of Real-Time and Embedded Systems) profile [6], published in 2009. Like the QoS/FT profile, the MARTE profile defines a general framework for describing quality of service aspects. The MARTE profile uses this framework to define a set of pre-made UML stereotypes, as those in the SPT profile.

The rest of this section presents the architecture of the MARTE specification and focuses on the key subset that has been used for the performance models.

A. Architecture

The MARTE profile is a complex specification, spanning over 700 pages. It is organised into several subprofiles and includes a normative model library with predefined types and concepts and an embedded expression language known as the Value Specification Language (VSL). Figure 1 lists each of the packages that constitute MARTE and their elements:

- The "MARTE foundations" package defines the core concepts that are used throughout the other profiles, such as the concept of a non-functional property (NFP) or how to model time, resources (using the General Resource Modelling or GRM subprofile) or the allocation of functional elements on the available resources.
- The "MARTE analysis model" package is used to annotate application models to support analysis of system properties. The Generic Quantitative Analysis Modelling

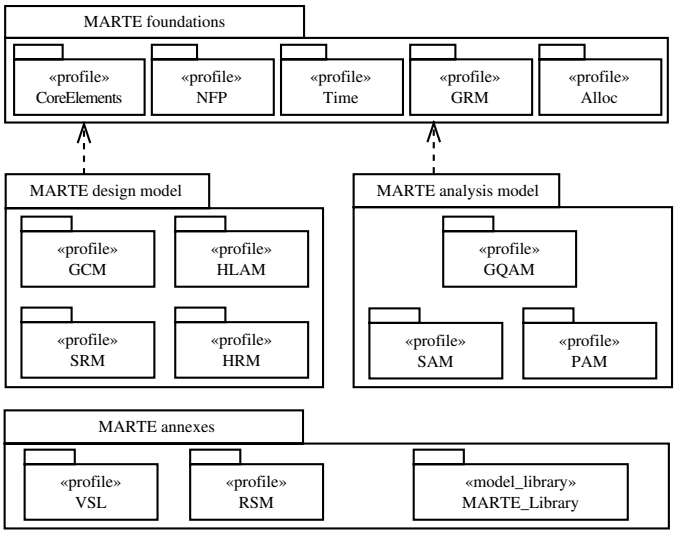

Fig. 1. Architecture of the MARTE profile [6]

(GQAM) subprofile uses the foundations package and the normative model library to provide a base set of concepts for the two kinds of analysis supported by MARTE: schedulability analysis and performance analysis. Schedulability analysis predicts whether a set of software tasks meets its timing constraints and is modelled using the Schedulability Analysis Modelling (SAM) subprofile. Performance analysis determines whether a system with non-deterministic behaviour can provide adequate performance, and is supported through the Performance Analysis Modelling (PAM) subprofile.

- The "MARTE design model" package provides the required concepts for modelling the features of real-time and embedded (RT/E) systems. The Generic Component Modelling (GCM) subprofile provides additional core concepts for RT/E systems. The High-Level Application Modelling (HLAM) subprofile provides the concept of a real-time execution unit that manages several resources and a queue of messages with various real-time requirements. Finally, the Detailed Resource Modelling (DRM) subprofile provides facilities for describing the software and hardware resources used by the system.
- The "MARTE annexes" package includes the Value Specification Language (VSL) used for all MARTE expressions, the Repetitive Structured Modelling (RSM) package for describing available software and hardware parallelism, and the normative MARTE model library.
  The normative MARTE library defines the set of standard primitive types (such as real numbers or integers) and derived types (such as vectors of integers or NFPs involving a real value), among many other concepts beyond the scope of this article.

B. GQAM

Using the Generic Quantitative Analysis Modelling (GQAM) subprofile requires the definition of an *AnalysisContext*, which is formed by a *WorkloadBehavior* object (the workload to be run) and a *ResourcesPlatform* object (the resources to be used). An *AnalysisContext* may also include a set of user-defined context parameters, which will be available as variables in the VSL expressions of the NFP.

The workload is then divided into the *WorkloadEvent* describing the request arrival pattern, and the *BehaviorScenario* specifying how these requests should be handled and the NFPs for them. A *BehaviorScenario* is further divided into *Step*s which are ordered using *PrecedenceRelation*s of several kinds, such as sequential, branching, merging, forking or joining relations. Each *Step* may have NFPs of its own. These NFPs include response time, throughput, utilisation or the expected number of repetitions.

Finally, the GQAM concepts are mapped to UML stereotypes. For instance, *AnalysisContext*, *BehaviourScenario* and *Step* are mapped to the ≪GaAnalysisContext≫, ≪GaScenario≫ and ≪GaStep≫ stereotypes, respectively.

C. VSL

As mentioned above, the GQAM *BehaviorScenario* and *Step* classes can contain NFPs for many aspects. However, properly describing the value of a NFP requires more than a simple scalar value: it is required to describe aspects such as measurement sources, measurement unit, precision and so on. In addition, the value of a NFP may be derived from a complex expression using several context parameters. All these features can be described using the Value Specification Language (VSL) embedded within the MARTE profile.

VSL provides a set of datatypes that extends the primitive types available in UML with composite types (such as intervals, collections or tuples) and subtypes. It also provides a textual syntax for complex expressions that may use conditional operators, invoke operators, compute time values and use arithmetic operators, among other features. Both can be combined: for instance, *(expr=2+3*f,ms,req)* is a VSL tuple that represents a duration in milliseconds (*ms*) that has been required by the developer (*req*) and is computed from the f context parameter as $2 + 3f$.

IV. Performance models

This section will present the notation used by the performance algorithms described in [2]. The models are used for performance analysis, and so PAM would appear to be the best starting point. However, the focus of the algorithms is different than the one favoured by PAM, which is predicting the performance of the whole system from its parts. Instead, the algorithms infer the performance needed in each part of the system from the global requirements. For this reason, it only uses the generic analysis core, the GQAM subprofile.

To keep the models simple, the notation only uses the three stereotypes in Section III-B. Due to the additional complexity in explicitly describing the precedence relations among the *Step*s, this information is inferred from the flows in the UML models.

Figure 2 shows a simple example. Inferred annotations are highlighted in bold:

1) The activity is annotated with a ≪GaScenario≫ stereotype, in which `respT` specifies that every request is completed within 1 second, and `throughput` specifies that 1 request per second needs to be handled. These expressions have their `source` attribute set to `req`, as they represent explicit requirements from the developer.
2) In addition, the activity declares a set of context parameters in the `contextParam` field of the ≪GaAnalysisContext≫ stereotype. These variables represent the time per unit of weight that must be allocated to their corresponding activity in addition to the minimum required time. Their values are computed by the time limit inference algorithm.
3) Each action in the activity is annotated with ≪GaStep≫, using in `hostDemand` a VSL expression of the form $m + ws$, where m is the minimum time limit, w is the weight of the action for distributing the remaining time, and s is the context parameter linked to that action. These expressions also have their `source` attribute set to `req`, for the same reasons as those in ≪GaScenario≫.
   The time limit inference algorithm adds a new constraint to `hostDemand`, indicating the exact time limit to be enforced. The throughput inference algorithm extends `throughput` with a constraint that lists how many requests per second should be handled. As these constraints have been automatically inferred, their `source` attribute is set to `calc` (calculated).
4) Outgoing edges from condition nodes also use ≪GaStep≫ but only for the `prob` attribute, which is set by the user to the estimated probability it is traversed.

V. Overall approach

The model shown in the previous section is entirely abstract: at that level of detail, it cannot be executed automatically. It will have to be implemented through other means.

After it has been implemented, it would be useful to take advantage of the original model to generate the performance test cases. However, the model lacks the required design and implementation details to produce executable artefacts. To solve this issue, several approaches could be considered:

1) The abstract model could be extended with additional information, but that would clutter it and make it harder to understand.
2) On the other hand, the implementation models could be annotated with performance requirements, but this would also pollute their original intent.
3) Finally, a separate model that links the abstract and concrete models could be used. This is commonly known as a *weaving model*. Several technologies already exist for implementing these, such as AMW [11] or Epsilon ModeLink [15]. While AMW uses a generic weaving metamodel, ModeLink is a more lightweight approach that requires defining custom weaving metamodels for every pair of metamodels.

In order to preserve the cohesiveness of the abstract performance model and the design and implementation models, the third approach has been chosen. The weaving model will need to allow users to annotate the links with the additional information required by the testing process, the target technologies and the generation process itself. With target technologies, we refer not only to the performance testing framework or tool which will run the generated tests, but also all the components which will be part of the test infrastructure. As we will see in Section VII, this may include IDEs (e.g. Eclipse) or build automation tools (Maven).

Some of the information may be shared by a set of tests (possibly all of them), and some of the information will be specific to a particular link between a design/implementation artefact and a performance requirement. For instance, while the number of threads used to exercise the system under test may need to be the same for all the tests, the interpretation of the time limit requirement as a median, an average or a percentile may change from test to test.

After establishing the required links, the next step is generating the tests themselves. To do so, a regular Model-to-Text (M2T) transformation could be used, written in a specialised language such as the Epsilon Generation Language [16]. In case it were necessary to slightly refine or validate the weaving model before, an intermediate Model-to-Model (M2M) transformation could be added. Figure 3 illustrates the models and steps involved in the overall approach.

In some cases, we may want to allow users to easily customise certain interesting parts of the tests, while abstracting them from the details that are less interesting. These interesting parts could be written into a custom domain-specific language instead of code, which would be interpreted as the tests were executed by augmenting the testing infrastructure accordingly. We will see an example of this with the TestSpec language later in Section VII-E.

The next sections will show two applications of the overall approach in Figure 3, using different technologies to assist in generating performance test artefacts in different environments. Both approaches have been implemented and are freely available under the open source Eclipse Public License at [17]. In order to develop these transformations, a bottom-up approach was used: a manually developed performance test environment was gradually replaced by automatically generated fragments until only the weaving model remained. After the entire process had been automated, the generators were refined to allow for more flexibility and convenience.

VI. Reusing Java unit tests as performance tests

Generating executable performance test cases from scratch automatically will usually require many detailed models and complex transformations, which are expensive to produce and maintain. The initial effort required may deter potential adopters. An alternative inexpensive approach is to repurpose existing functional tests as performance tests as a starting

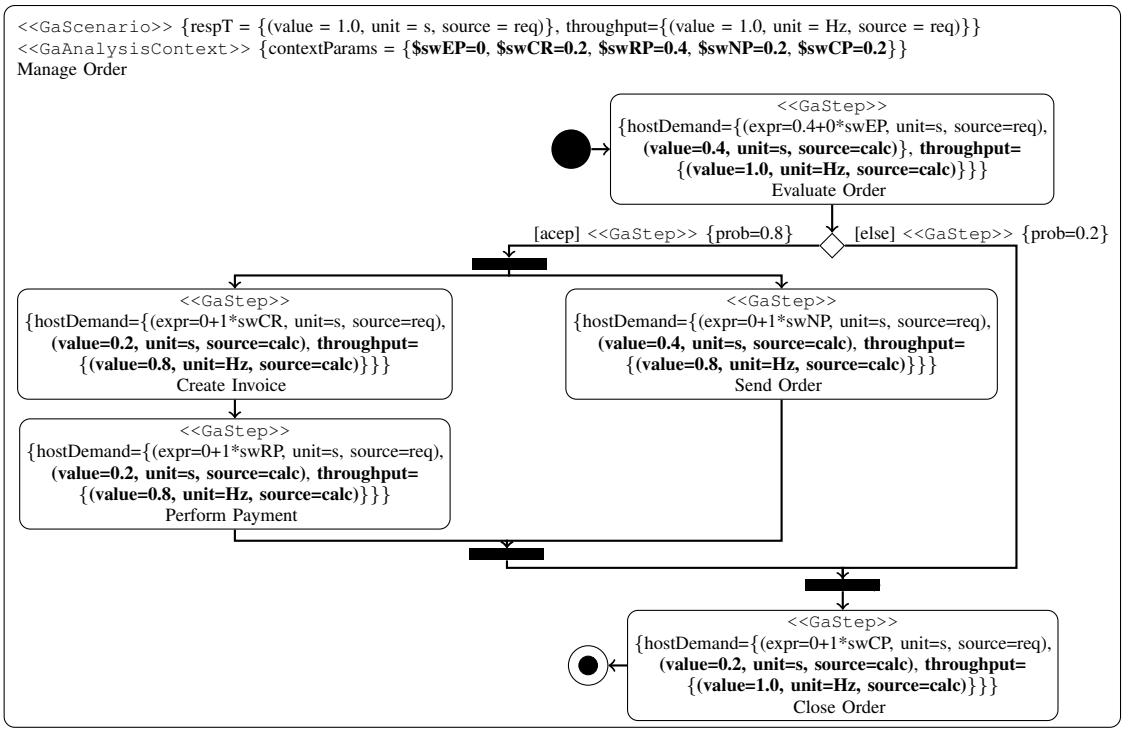

Fig. 2. Simple example model annotated by the performance inference algorithms

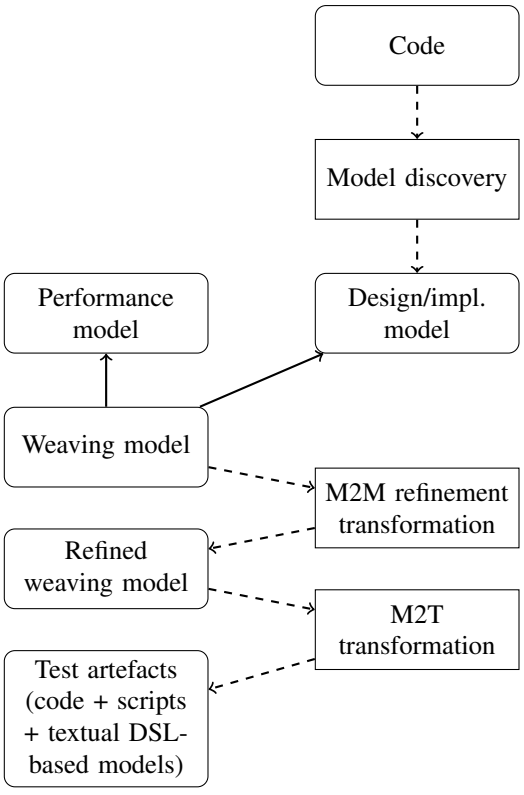

Fig. 3. Overall approach for generating performance test artefacts from abstract performance models

```
@RunWith(ContiPerfSuiteRunner.class)
@SuiteClasses(TFunctionalJUnit4.class)
@PerfTest(invocations = 100, threads = 10)
@Required(max=1000)
public class InferredLoadTest {}
```

Listing 1. Java code for wrapping the *TFunctionalJUnit4* JUnit 4 test suite using ContiPerf

point. This is the aim of libraries such as ContiPerf [18]. The rest of the section will show the overall approach in Figure 3 was customised for this particular use case. The resulting transformation chain is shown in Figure 4.

A. Target framework: ContiPerf

Listing 1 shows how ContiPerf is normally used. Instead of using Java objects, ContiPerf uses Java 6 annotations, which are easier to generate automatically. The *@PerfTest* annotation indicates that the test will be run 100 times using 10 threads, so each thread will perform 10 invocations. *@Required* indicates that each of these invocations should finish within 1000 milliseconds at most. *@SuiteClasses* points to the JUnit 4 test suites to be reused for performance testing, and *@RunWith* tells JUnit 4 to use the ContiPerf test runner.

B. Model extraction

In both cases, the code itself is straightforward to generate. However, the generated code must integrate correctly with the existing code. If the code was not produced using a model-driven approach, there will not be a design or implementation

model to link to. Instead, a model of the structure of the existing code is derived using the Eclipse MoDisco model discovery tool [19]. Eclipse MoDisco can generate models from Java code such as that shown in Figure 5.

C. Weaving metamodel

Once the performance and the implementation models have been produced, the next step is to link them using a new *weaving model* that conforms to the metamodel in Figure 6. Some of the types in the weaving metamodel refer to types in the *uml* and *java* packages from the UML2 metamodel and the MoDisco Java metamodel, respectively.

Each model consists of an instance of *PerformanceRequirementLinks*, which provides several global configuration parameters and contains a set of *PerformanceRequirementLink* instances. Users can set the number of samples which should be collected for each test, the number of threads over which these should be distributed and the directory under which the code should be generated. Every link relates an UML *ExecutableNode* with a Java class: if no *MethodDeclaration*s are specified, all tests will be reused. Otherwise, only the selected methods will be reused. Finally, the target time limit may be enforced as a maximum value (`MAX`), average (`AVERAGE`), median (`MEDIAN`) or a percentile (the rest).

Originally, the models referenced the MARTE ≪GaStep≫ stereotype instead of the UML *ExecutableNode*. These references were switched to *ExecutableNode* as the ≪GaStep≫ stereotype was optional if the default minimum time limit $m = 0$ and weight $w = 1$ were used.

D. Code generation

Models are populated by combining the standard Epsilon Modeling Framework (EMF) tree-based editors and the three-pane Epsilon ModeLink editor (as in Figure 7). ModeLink provides a drag-and-drop approach to model linking that is convenient for model weaving. The EMF editors have been manually customised so users may only pick JUnit 4 test suites and test methods.

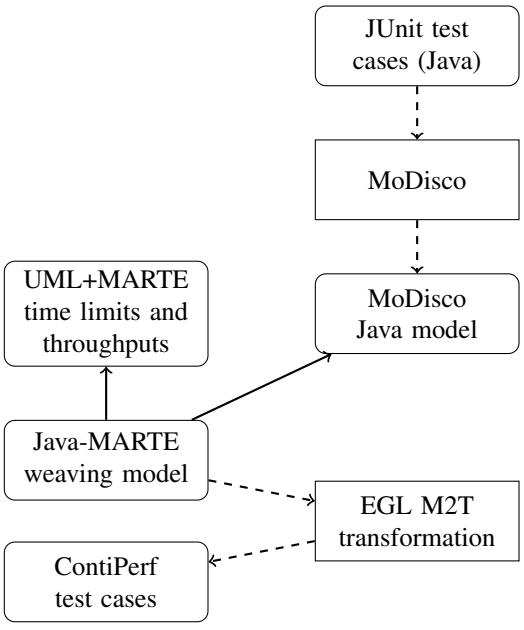

Fig. 4. Instance of the overall approach for wrapping JUnit tests into ContiPerf tests

```
@Required(throughput=2, max=400)
public class WrapSomeTests extends OriginalSuite {
  @Rule public MethodRule f =
    new FilterByClassRule(this.getClass());

  @Rule public ContiPerfRule i = new ContiPerfRule();

  @PerfTest(invocations=1000, threads=5)
  @Test @Override
  public void first() throws Exception {
    super.first();
  }

// protected region customTests off begin
// Add your own tests here
// protected region customTests end
}
```

Listing 2. Java code wrapping one test from *OriginalSuite* using ContiPerf

```
@WebService
public class HelloWorld {
  @WebMethod
  public String greet(
    @WebParam(name="name") String name)
  {
    return "Hello " + name;
  }
}
```

Listing 3. Java code using JAX-WS for a "HelloWorld" Web Service

The code is generated using a set of Epsilon Generation Language (EGL) templates. When all tests are reused as performance tests, the generated code will use the *ContiPerfSuiteRunner* test runner, as in Listing 1.

However, when only some tests are wrapped the code will resemble that in Listing 2. The *ContiPerfRule* would normally convert all tests into performance tests. By using the *FilterByClassRule* helper class (also generated with EGL), the generated code will be able to specify that only some of those tests need to be reused as performance tests.

VII. Generating performance tests for WSDL-based Web Services

In the previous section, the approach was applied to existing JUnit test cases, repurposing them as performance test cases. This section will discuss how to generate performance test artefacts for a Web Service (WS) [20] in a language agnostic manner. The implemented solution is summarised in Figure 8.

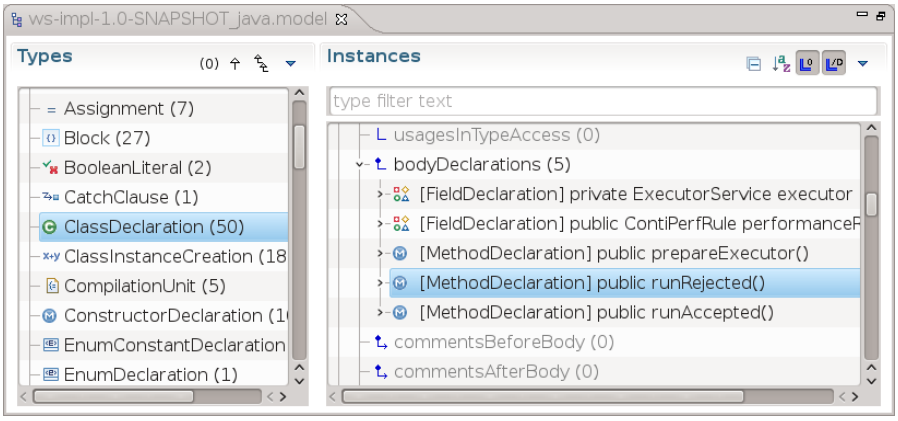

Fig. 5. MoDisco model browser showing a model generated from an Eclipse Java project

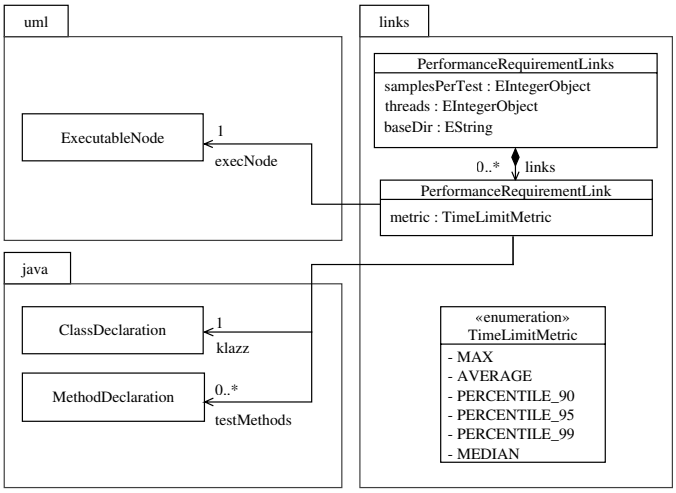

Fig. 6. Java-MARTE weaving metamodel

A. Motivation

Web Services based on the WS-* technology stack are usually described using a Web Services Description Language (WSDL) [21] document. This XML-based document is an abstract and language-independent description of the available operations for the service and the messages to be exchanged between the service and its consumers.

Existing Web Service frameworks such as Apache CXF [22] can generate most of the code required to implement and consume the services from the WSDL document. Users only need to implement the business logic of the services. In addition, some frameworks (CXF included) can work in reverse, generating WSDL from adequately annotated code.

Listing 3 shows an example fragment of Java code that implements a simple “HelloWorld” Web service using standard JAX-WS [23] annotations. This Java code could be tested using the approach in Section VI. However, a WSDL-based approach would be easier to work with when mixing services written in multiple languages or frameworks.

B. Target performance testing tool: The Grinder

The previous section reused unit tests written in a particular language (Java) and a particular framework (JUnit). Therefore, the target technology was an extension upon this framework (ContiPerf). However, since the WSDL description of a Web Service does not depend on the language that it is implemented in, we are not limited to a specific language for the tests. Instead, we will use a dedicated performance testing tool. Such tools help define tests with less cost and in a way that is independent of the implementation language of the software under test.

We evaluated the following tools based on the ease with which test specifications could be generated for them, by developing a simple performance test on a single service with each of them and studying the files required by the tools:

- The Grinder [24] used textual configuration files to configure the test environment, which executes Jython scripts that use the public API provided by the tool.
- Apache JMeter [25] used reflective XML documents. Most of their contents were directly translated into API calls of the underlying Java code, tightly coupling the transformation to their internal code structure.
- Eviware loadUI [26] had the most complicated input format out of the three. It used both binary and textual artefacts. Some of the textual artefacts were trees of Java classes, which would have to be generated and then packed together with the binary parts.

The Grinder [24] was selected among the available tools, as its input format was the easiest to generate and provided more flexibility.

In addition, The Grinder is easy to scale up depending on the testing requirements. The Grinder can launch several processes that spawn a certain number of threads which will repeatedly run the test. It can also optionally distribute work over several machines: one of them provides a graphical console and acts as the master, and the rest are agents that manage a set of worker processes.

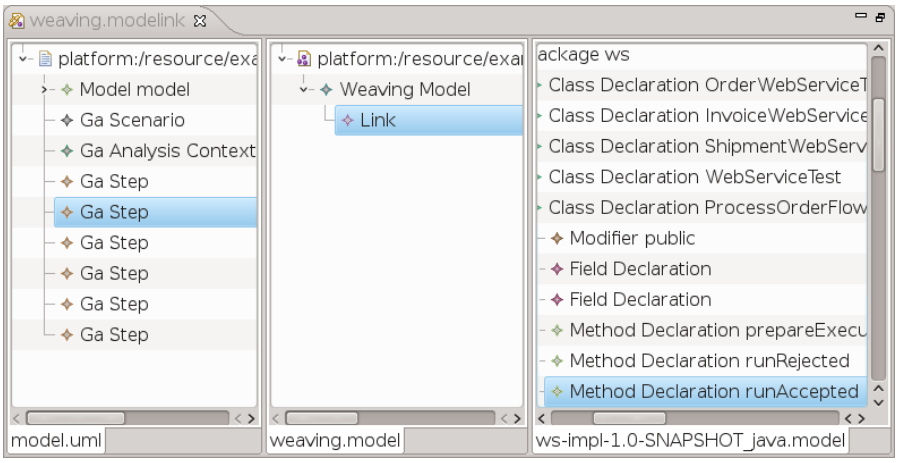

Fig. 7. Screenshot of the Epsilon ModeLink editor weaving the MARTE performance model and the MoDisco model

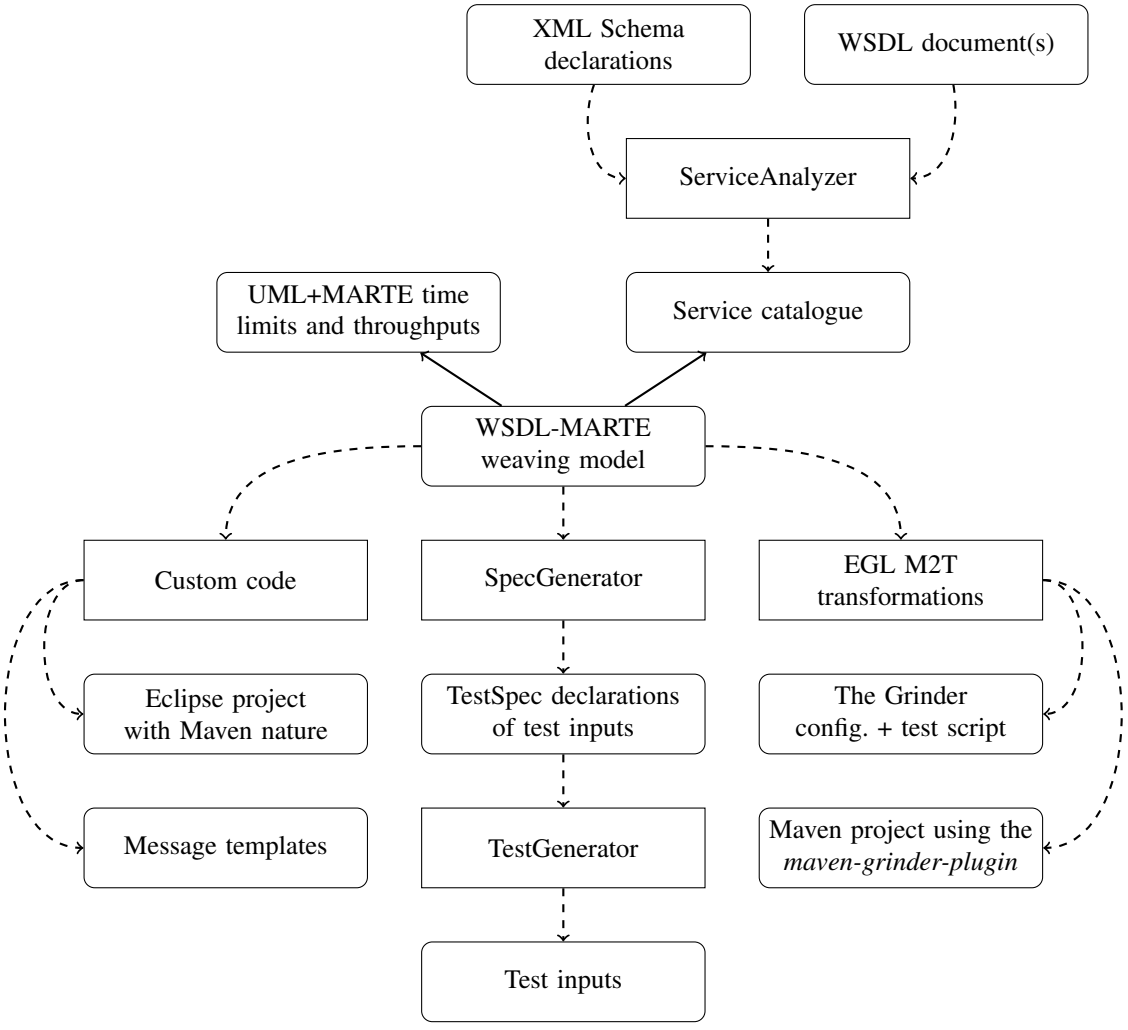

Fig. 8. Instance of the overall approach for generating performance tests for WSDL-based Web Services. In comparison with the approach specifically targeted for Java, this approach requires integrating several technologies, such as a build automation tool (Maven), three custom tools (ServiceAnalyzer, SpecGenerator and TestGenerator) and a dedicated performance testing tool (The Grinder), among others.

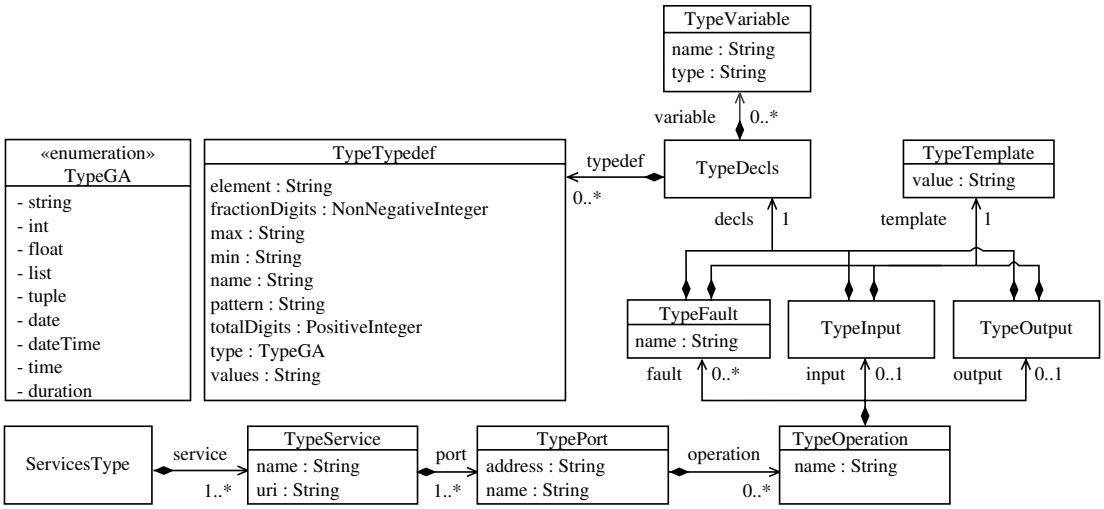

Fig. 9. ServiceAnalyzer service catalogue metamodel

C. Model extraction

Since WSDL documents are declarative and language-independent descriptions of the Web Services, the original proposal intended to use them as design models. After transforming automatically the XML Schema description of the WSDL document format into a regular ECore metamodel [27], WSDL documents would be loaded as regular Eclipse Modeling Framework models, reusing most of the technologies mentioned in Section VI.

In practice, however, WSDL documents are too complex to be used as-is for model weaving and model transformation. WSDL documents can be divided across multiple files and machines and combine descriptions in the WSDL and XML Schema formats. In addition, XML Schema and WSDL are highly flexible, allowing many possibilities that may or may not be implemented by vendors. This has led to the definition of specifications such as the Web Services Interoperability Basic Profile (WS-I BP) [28], which restricts these standards to a consistent subset that is well implemented across vendors.

Therefore, it was decided to extract models from the WSDL documents themselves using a new custom tool, ServiceAnalyzer, also available as open source from [17]. ServiceAnalyzer produces a "service catalogue" from a set of local or remote WSDL documents that conform to the WS-I BP. Service catalogues can be loaded as an EMF model by using their XML Schema definition, as originally intended for WSDL.

The service catalogue metamodel is shown in Figure 9. Models are instances of *ServiceType*, which contains a set of *TypeService*s with their own *TypePort*s. Each *TypePort* has a collection of *TypeOperation*s that may have an input, an output, and/or several fault messages. Message descriptions are divided into a *TypeTemplate* containing an Apache Velocity [29] template, and a *TypeDecls* that declares the variables used within the Velocity template. Variables may belong to one of the predefined types in *TypeGA*, which are based on the XML Schema primitive types, or they may belong to a custom type defined with a *TypeTypedef*.

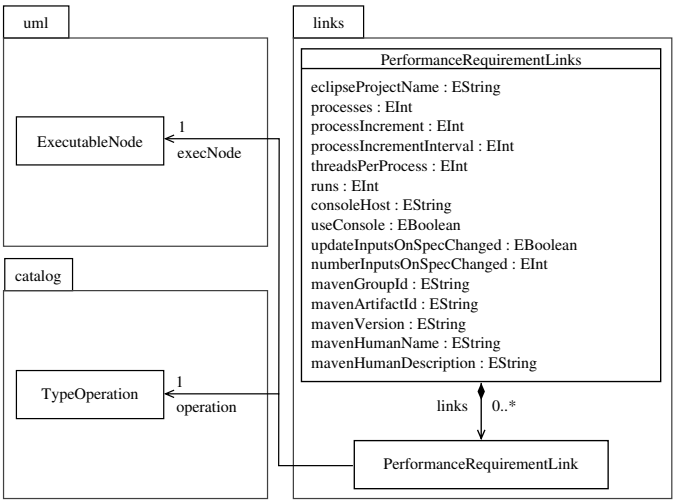

Fig. 10. ServiceAnalyzer-MARTE weaving metamodel

Services store their names and namespace URIs, ports store their names and the URLs they are listening at, and operations and faults store their names. Type definitions must specify at least a name and a base type, but they usually specify additional restrictions such as a pattern based on a regular expression (`pattern`), minimum or maximum values (`min` or `max`) or a set of accepted `values`, among others.

D. Weaving metamodel

The weaving model needs to relate the *ExecutableNode*s in the UML activity diagram with the *TypeOperation*s in the ServiceAnalyzer service catalogue. For instance, a developer might want to ensure that every invocation of the *evaluate* operation of the *Order* service finishes within a certain time while handling a certain number of requests per second.

The weaving metamodel is shown in Figure 10. It is quite similar to that in Figure 9, but the global options in the

PerformanceRequirementLinks class have been changed to reflect the target technologies for this transformation:

- `eclipseProjectName` is the name of the Eclipse project which will be generated by the transformer. By default, it is set to "performance.tests".
- The attributes ranging from `process` to `useConsole` are directly mapped to the configuration options of The Grinder with the same name. `process` is the number of worker processes that will be used by each agent, starting from 1 and increasing by `processIncrement` every `processIncrementInterval` milliseconds (by default, by 1 every second). Each worker process will spawn as many as `threadsPerProcess` threads and repeat the tests the number of times indicated in `runs`. If `useConsole` is set to "true", the console process at `consoleHost` will distribute work over the agents connected to it.
- The rest of the attributes can be used to customise the metadata of the Maven project that is generated by the transformer.

As for the options for the testing process itself, `updateInputsOnSpecChanged` and `numberInputsOnSpecChanged` indicate if the test inputs should be updated when the `.spec` file describing their format changes, and how many should be generated each time.

The default options should be good enough for most users. In the next sections, we will mention again some of them as we introduce the following steps in the generation process.

E. Test data generation

In order to run performance tests, it is necessary to provide them with test inputs so they can exercise the WS appropriately. Doing this in a completely automated way is outside the scope of the approach. As an initial approximation, data inputs are randomly generated using uniform distributions, based on the variables and templates in the service catalogue.

First, the tools extract the appropriate Velocity templates and variable declarations from the ServiceAnalyzer service catalogue to separate files.

Listing 4 shows an Apache Velocity template which can produce every valid request for an order evaluation service, according to its WSDL and XML Schema declarations. As a template language, the Velocity language is kept simple, providing only the most common programming constructs, such as conditionals (`#if`), loops over a list (`#foreach`), variable assignments (`#set`) or field references (`$var.field`). Velocity templates are expanded during test execution with the variables loaded into their *contexts*. This template produces a `<newOrder>` element for each item in `$evaluate`. In turn, the template produces a `<articleQuantities>` element for each item, with the appropriate article identifier and requested quantities.

Listing 5 shows the TestSpec declarations that were extracted from the same catalogue entry. The TestSpec language is implemented by the TestGenerator tool, also available from [17]. It is a simple domain-specific language (inspired on C declarations) which allows users to define new scalar, list and tuple types based on a set of primitive types based on XML Schema. These new types can have additional constraints, such as having minimum or maximum values or lengths, adhering to a certain regular expression or having a certain number of digits. From these declarations, TestGenerator can produce an arbitrary number of random tests and store them as Velocity templates.

The Velocity files produced by TestGenerator set up the context to be used to generate the message templates. They consist of a sequence of variable assignments in which every variable receives a list of values to be used within each test. Listing 6 shows three test cases that were produced from

```
<w:evaluate xmlns:w="http://ws.sodmt.uca.es/">
  #foreach($V1 in $evaluate)
  <newOrder>
    #foreach($V2 in $V1)
    <articleQuantities>
      <articleID>
        $V2.get(0)
      </articleID>
      #foreach($V3 in $V2.get(1))
      <quantity>
        $V3
      </quantity>
      #end
    </articleQuantities>
    #end
  </newOrder>
  #end
</w:evaluate>
```

Listing 4. Apache Velocity template extracted from the ServiceAnalyzer catalog for producing the body of a message from test data

```
typedef int (min=0, max=100) TArtID;
typedef float (min=0.01, max=2000) TPrice;
typedef list (element=TPrice, min=1, max=1) TL_float;
typedef tuple (element={TArtID, TL_float}) TArticleQtys;
typedef list (element=TArticleQtys, min=0) TOrder;
typedef list (element=TOrder, min=1, max=1) TEvaluate;
TEvaluate evaluate;
```

Listing 5. TestGenerator `.spec` extracted from the ServiceAnalyzer catalog describing the test data for the template in Listing 4

```
#set($evaluate = [
  [[[85, [1530.1414]], [3, [1652.419]], [50, [550.96515]]]],
  [[[92, [1682.8262]], [45, [1593.5898]]]],
  [[[79, [72.64899]], [22, [603.8968]], [8, [1278.9677]]]]
])
```

Listing 6. Test data produced by TestGenerator from the `.spec` in Listing 5

```
grinder.processes=5
grinder.runs=100
grinder.processIncrement=1
grinder.processIncrementInterval=1000
```

Listing 7. Example `.properties` file with configuration parameters for the workload

```
class TestRunner:
  def __call__(self):
    def invoke():
      response = HTTPRequest().POST(
        "http://localhost:8080/orders",
        "(... SOAP message ...)")
      stats = grinder.statistics.getForCurrentTest()
      stats.success = (response.statusCode != 200
                       and stats.time < 150)
      test = Test(1, "Query order by ID").wrap(invoke)
      test()
```

Listing 8. Example Jython script for The Grinder with the contents of the performance test to be run by each simulated client

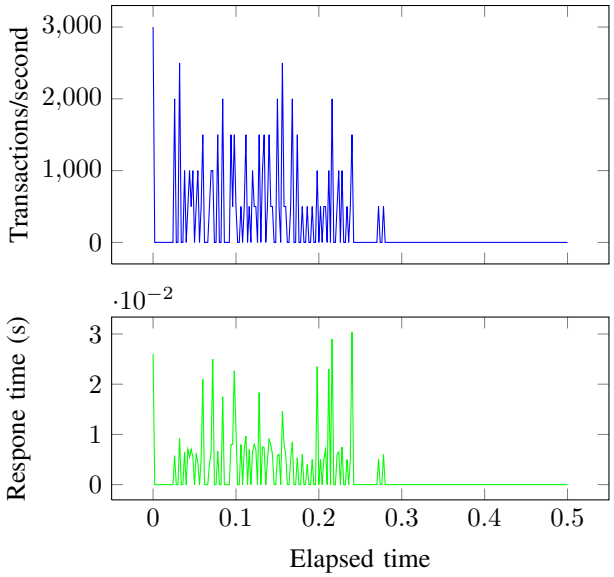

Fig. 11. Overall performance graph produced by Grinder Analyzer

the `.spec` in Listing 5. For instance, the first test requests 1530.14 units of article #85, 1652.419 units of article #3 and 550.965 units of article #50. We used Velocity to store test data since it was more flexible than a simple table or spreadsheet, as it allowed for arbitrarily nested lists.

In the wild, WSDL declarations tend to be quite lax, allowing messages with no upper bound on their length or elements containing negative integers, even though they are not accepted. In these cases, users may want to customise the service catalogue before generating the `.spec` descriptions from it. This will change the values used for all tests of the modified operations. Alternatively, users may want to modify a single `.spec` file describing the inputs of a particular test. Users may also customise the message templates with additional logic, or provide manually designed input data instead of generating random inputs.

The explicit separation of the service interface, message generation template, test generation specification and test data provides a great deal of flexibility. Later iterations of this application could generate larger parts of the test plan by implementing more advanced test generation strategies beyond random generation. These advanced strategies could be expressed as part of the links in the weaving model. The strategy could be applied in the weaving model refining step showed in Figure 3.

F. Test code generation

After weaving the service catalogue model with the MARTE model and producing some input data to exercise the Web Services, the next step is generating the test specification for The Grinder.

The Grinder requires generating two different files: a `.properties` file indicating several parameters of the workload to be generated, and a Jython script with the test to be run by each simulated client. Listings 7 and 8 show simplified examples for these two files. These files are automatically generated using EGL.

The `.properties` file in Listing 7 indicates that 5 processes should each run the test 100 times, starting with 1 process and adding one more every 1000 milliseconds. On the other hand, the test itself consists of sending an appropriate SOAP message to a specific URL and checking that the response has the OK (200) HTTP status code and that it was received within 150 milliseconds. These values are extracted from the global options in the *PerformanceRequirementLinks* object of the model. `consoleHost` and `useConsole` are also used in the `.properties` file.

The actual generated Jython script is over 180 lines long and takes advantage of several language features to avoid code repetition. In addition to running the tests themselves, it can regenerate test data if the `.spec` files have been customised by the user since the last run. Every time a test is run, a set of input values is randomly selected from the available test data. This input data is used to generate the SOAP message from the message templates, invoke the service and check the non-functional attributes of the reply. One limitation with the current version of the scripts is they can only check maximum response times, unlike the approach in Section VI, which can handle averages, medians and percentiles.

G. Test infrastructure and report generation

The approach in Section VI was straightforward: as it simply produced Java code based on the ContiPerf library, users would simply need to add ContiPerf to their development environments and the run the tests using standard tools. However, running the tests produced by this approach would require setting up TestGenerator, The Grinder, and Apache Velocity.

Operation	Passed tests	Failed tests	Bytes per second	Mean response length
Close Order	60	0	66,590	332.95
Evaluate Order	60	0	64,333.33	321.67

TABLE I
TEST METRICS PRODUCED BY GRINDER ANALYZER (OVERALL RESULTS, THROUGHPUT AND MESSAGE SIZES)

Operation	Mean response time	Response time std. dev.	Mean time DNS	Mean time conn.	Mean time first byte
Close Order	14.35	15.62	0	0.13	13.18
Evaluate Order	8.98	6.01	0	0.37	5.93

TABLE II
TEST METRICS PRODUCED BY GRINDER ANALYZER (TIMING INFORMATION)

For this reason, the tools implement an additional EGL transformation that produces an Apache Maven [30] project description that automatically downloads all dependencies, runs the performance tests and produces test reports from the results. The Grinder is integrated through the open source plug-in available at [31]. Maven also enforces a standard directory layout for all the generated artefacts.

This infrastructure allows users to run the entire testing process with a single `mvn post-integration-test` command, which also invokes the Grinder Analyzer tool [32] on the raw logs to produce an HTML report including (but not limited to) the information shown in Figure 11 and Tables I and II. The report includes both a graph with the response times and transactions per seconds obtained, and a table with more detailed information. The report shows that all tests passed and that the mean response time for the tested service was 8.98 milliseconds. These results are to be expected, since the tool was tested against local Web Services using an in-memory object-relational database. Applying this approach to real-world WS is a future line of work.

VIII. CONCLUSION AND FUTURE WORK

This work has described an overall approach for generating performance test artefacts from the abstract performance models produced by the inference algorithms in [2]. To generate concrete test artefacts while keeping the abstract performance models separated from any design or implementation details, the approach links the performance model to a design or implementation model using an intermediate *weaving model*. If a design or implementation model is not available, it can be extracted from the existing code. The weaving model can be then optionally refined using a model-to-model transformation, and finally transformed into the performance test artefacts with a model-to-text transformation.

The general approach has been validated by applying it on two target technologies. Both approaches have been successfully implemented and are freely available under the open source Eclipse Public License at [17].

The first application weaves JUnit test suites with MARTE models and converts all or some of their unit tests into performance test cases, using the ContiPerf library. The implementation model is extracted from the Java code implementing the test cases using the model discovery tool MoDisco [19], and the weaving model links the *ExecutableNode*s in the UML activity diagram to the Java tests in the MoDisco model.

The second application can generate performance test cases for any Web Service that is described using the WSDL specification [21]. It is independent of the language in which the Web Service has been implemented, as it is based on a special-purpose performance testing tool: The Grinder [24]. Users extract service catalogues from a set of WSDL documents and then weave the service operations in the catalogue with the MARTE models. The service catalogues also include message templates and template variable declarations, which are used to randomly generate a set of initial test inputs. Users are able to manually customise the service catalogue, the message templates and the test inputs. In addition to the inputs, a set of automated model-to-text transformation produces the Jython code and the configuration file required by the Grinder, and a Maven project description that enables users to run the tests and produce reports with a single command (as those in Section VII-G).

While these applications show that the overall approach can be reused for different target technologies, they do currently share several limitations. Transformations only know the part of the system under test that is strictly needed to generate the tests. For this reason, users will need to manually customise the tests if they need to restore the state of the system after a performance test, a memory violation or an aborted operating system process, or if they want to set up specific mockups for specific subsystems in the application. Nevertheless, the transformations could assist the user by providing clear "hooks" where this kind of logic could be placed, and keeping those "hooks" from being overwritten if the tests are generated. This is already being done in the Java approach: the generated test suites use EGL protected areas that are preserved when the files are regenerated.

Future lines of work include:

- Continue evaluating both approaches by applying them to larger Web Services running in remote services and using larger data sets. The present versions were developed using local WS with small in-memory databases. One of the case studies under consideration is the Worldtravel testbed in [33], which implements a working business application backed by a relational database with more than 1GB of data.
- Enhance the Jython code generated in the second approach to include the same target metrics as in ContiPerf, such as average time, median time or percentiles. Usually, Service Level Agreements are defined in terms of percentages ("90% of the requests should be attended in x seconds").

- Provide more advanced strategies for generating test inputs for the WSDL-based WS. Currently, all inputs are generated using a uniform random distribution, but other random distributions could be combined. Alternatively, an evolutionary algorithm could be used to look for test cases that produce SLA violations, as proposed by Di Penta et al. in [9].
- Handle more complex service level agreements beyond meeting a certain service level objective (throughput and/or response times in our case). For instance, the current algorithms do not take into account the fact that a developer may want to enforce different SLOs than a customer in production. These particular variations could be handled by the weaving model itself, however, by adding appropriate global options to scale back the performance requirements.
- Evaluate the overall approach for other target technologies, such as unit tests written for other programming languages or different kinds of systems altogether, such as multimedia applications or graphical user interfaces.

ACKNOWLEDGEMENTS

This work was funded by the research scholarship PU-EPIF-FPI-C 2010-065 of the University of Cádiz, the MoDSOA project (TIN2011-27242) under the National Program for Research, Development and Innovation of the Ministry of Science and Innovation (Spain) and by the PR2011-004 project under the Research Promotion Plan of the University of Cádiz.

REFERENCES

[1] A. García-Domnguez, I. Medina-Bulo, and M. Marcos-Bárcena, "An approach for performance test artefact generation for multiple technologies from MARTE-Annotated workflows," in *7th International Conference on Internet and Web Applications and Services (ICIW 2012)*, Stuttgart, Germany, June 2012.

[2] A. García-Domínguez, I. Medina-Bulo, and M. Marcos-Bárcena, "Model-driven design of performance requirements with UML and MARTE," in *Proceedings of the 6th International Conference on Software and Data Technologies*, vol. 2. Seville, Spain: SciTePress, July 2011, pp. 54–63.

[3] M. Woodside, G. Franks, and D. Petriu, "The future of software performance engineering," in *Proc. of Future of Software Engineering 2007*, 2007, pp. 171–187.

[4] D. C. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications," in *Proc. of the 12th Int. Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 2002)*, ser. Lecture Notes in Computer Science. London, UK: Springer Berlin, 2002, vol. 2324, pp. 159—177.

[5] M. Tribastone and S. Gilmore, "Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile," in *Proc. of the 7th Int. Workshop on Software and Performance*. Princeton, NJ, USA: ACM, 2008, pp. 67–78.

[6] Object Management Group, "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) 1.0," http://www.omg.org/spec/MARTE/1.0/, November 2009, last checked on 2012-03-03.

[7] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing," Working Paper 04/2006, April 2006, last checked on 2012-12-20. [Online]. Available: http://researchcommons.waikato.ac.nz/handle/10289/81

[8] C. Barna, M. Litoiu, and H. Ghanbari, "Model-based performance testing (NIER track)," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 872–875.

[9] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, "Search-based testing of service level agreements," in *Proceedings of Genetic and Evolutionary Computation Conference*, H. Lipson, Ed. London, United Kingdom: ACM, July 2007, pp. 1090–1097.

[10] K. Suzuki, T. Higashino, A. Ulrich, T. Hasegawa, A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini, "Model-based generation of testbeds for web services," in *Testing of Software and Communicating Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5047, pp. 266–282.

[11] M. D. Del Fabro, J. Bézivin, and P. Valduriez, "Weaving models with the Eclipse AMW plugin," in *Proceedings of the 2006 Eclipse Modeling Symposium, Eclipse Summit Europe*, Esslingen, Germany, October 2006.

[12] J. M. Vara, M. V. De Castro, M. Didonet Del Fabro, and E. Marcos, "Using weaving models to automate model-driven web engineering proposals," *International Journal of Computer Applications in Technology*, vol. 39, no. 4, pp. 245–252, 2010.

[13] OMG, "UML Profile for Schedulability, Performance, and Time (SPTP) 1.1," January 2005, last checked on 2012-12-20. [Online]. Available: http://www.omg.org/spec/SPTP/1.1/

[14] ——, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QFTP) 1.1," http://www.omg.org/spec/QFTP/1.1/, April 2008.

[15] D. S. Kolovos, "Epsilon ModeLink," 2010, last checked on 2012-12-20. [Online]. Available: http://eclipse.org/epsilon/doc/modelink/

[16] D. S. Kolovos, R. F. Paige, L. M. Rose, and A. García-Domínguez, "The Epsilon Book," 2011, last checked on 2012-12-20. [Online]. Available: http://www.eclipse.org/epsilon/doc/book

[17] A. García-Domínguez, "Homepage of the SODM+T project," January 2012, last checked on 2012-12-20. [Online]. Available: https://neptuno.uca.es/redmine/projects/sodmt

[18] V. Bergmann, "ContiPerf 2," September 2011, last checked on 2012-12-20. [Online]. Available: http://databene.org/contiperf.html

[19] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Antwerp, Belgium, September 2010, pp. 173–174.

[20] H. Haas and A. Brown, "Web services glossary," World Wide Web Consortium, W3C Working Group Note, February 2004, last checked on 2012-12-20. [Online]. Available: http://www.w3.org/TR/ws-gloss/

[21] World Wide Web Consortium, "WSDL 2.0 part 1: Core Language," June 2007, last checked on 2012-12-20. [Online]. Available: http://www.w3.org/TR/wsdl20

[22] Apache Software Foundation, "Apache CXF," November 2012, last checked on 2012-12-20. [Online]. Available: https://cxf.apache.org/

[23] Java.net, "JAX-WS reference implementation," November 2011, last checked on 2012-12-20. [Online]. Available: http://jax-ws.java.net/

[24] P. Aston and C. Fizgerald, "The Grinder, a Java Load Testing Framework," 2012, last checked on 2012-12-20. [Online]. Available: http://grinder.sourceforge.net/

[25] Apache Software Foundation, "Apache JMeter," November 2011, last checked on 2012-12-20. [Online]. Available: http://jakarta.apache.org/jmeter/

[26] eviware.com, "loadUI homepage," 2012, last checked on 2012-12-20. [Online]. Available: http://www.loadui.org/

[27] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed., ser. Eclipse Series. Addison-Wesley Professional, December 2008.

[28] Web Services Interoperability Organization, "Basic profile - version 1.1 (Final)," April 2006, last checked on 2012-12-20. [Online]. Available: http://www.ws-i.org/Profiles/BasicProfile-1.1.html

[29] Apache Software Foundation, "Apache Velocity Project homepage," November 2010, last checked on 2012-12-20. [Online]. Available: http://velocity.apache.org

[30] ——, "Apache Maven homepage," January 2012, last checked on 2012-12-20. [Online]. Available: http://maven.apache.org

[31] G. Iacono and F. Muñoz-Castillo, "grinder-maven-plugin homepage," July 2012. [Online]. Available: http://code.google.com/p/grinder-maven-plugin/

[32] T. Bear, "Grinder Analyzer homepage," July 2012, last checked on 2012-12-20. [Online]. Available: http://track.sourceforge.net/

[33] P. Budny, S. Govindharaj, and K. Schwan, "Worldtrael: A testbed for service-oriented applications," *Service-Oriented ComputingICSOC 2008*, pp. 438–452, 2008.